공학석사 학위논문

MapReduce 환경에서 LSH 기반의 K-NN 그래프 생성 알고리즘의 개선

An Improvement in K-NN Graph Construction with Locality Sensitive Hashing on MapReduce

2015 년 2 월

서울대학교 대학원

전기·컴퓨터공학부

이 인 회

초 록

k-Nearest Neighbor(k-NN)그래프는 모든 노드에 대한 k-NN 정보를 나타내는 데이터 구조로써, 많은 정보검색 및 추천 시스템에서 k-NN그래프를 활용하고 있다. 이러한 장점에도 불구하고 brute-force방법의 k-NN 그래프 생성 방법은 $O(n^2)$의 시간복잡도를 갖기 때문에 빅데이터 셋에 대해서는 처리가 곤란하다. 따라서, 고차원, 희소 데이터에 효율적인 Locality Sensitive Hashing 기법을 분산환경인 MapReduce 환경에서 사용하여 k-NN그래프를 생성하는 알고리즘이 연구되고 있다. K-NN 그래프 생성은 사용자를 이웃후보 그룹으로 만들고 후보그룹 내의 쌍에 대해서만 brute-force하게 유사도를 계산하는 divide-and-conquer(two-stage) 방법을 사용했다. 특히, 그래프 생성과정 중 유사도 계산하는 부분이 가장 많은 시간이 소요되므로 후보 그룹을 어떻게 만드는 것인지가 중요하다. 기존의 방법은 사이즈가 큰 후보그룹을 방지하는데 한계점이 있다. 본 논문에서는 효율적인 k-NN 그래프 생성을 위하여 사이즈가 큰 후보그룹을 hierarchical LSH를 사용하여 재구성하는 알고리즘을 제시하였다. 실험 결과 본 논문에서 제시한 방법은 기존의 방법보다 빠르게 더 정확한 근사 그래프를 생성함을 확인하였다.

주요어 : 빅데이터, 맵리듀스, k-NN그래프 생성, LSH, MinHash

학 번 : 2013-20855

목 차

표 목 차

그 림 목 차

제 1 장

서론

k-Nearest Neighbor(k-NN)그래프는 모든 노드(node)에 대한 k-NN 정보를 나타내는 데이터 구조이다. k-NN그래프는 한 노드로부터 k개의 진출간선(outgoing edge)이 있는 그래프로써, 간선은 그 노드로부터 k-NN의 관계에 있다는 것을 알려준다. 그림 1은 2-NN그래프의 예를 보여주고 있다.

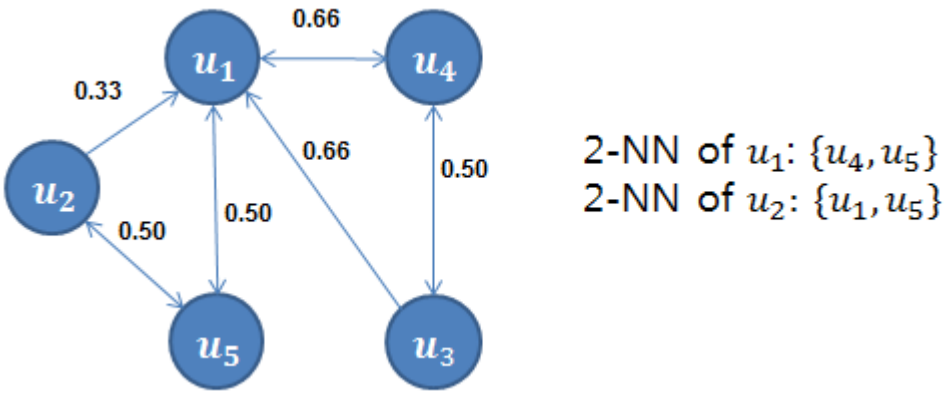


그림 1 2-NN 그래프의 예

이 그래프에서 각 노드의 진출간선은 그 노드의 2-NN을 나타내며, 각 진출간선에 대응하는 값은 2-NN에 대한 유사도를 나타낸다. 예를 들어 u_1의 2-NN은 u_1으로부터의 진출간선을 따라가면 $\{u_4, u_5\}$을 찾을 수 있다.

본 논문의 목표 고차원(high-dimension)의 데이터로부터 효율적으로 k-NN그래프를 생성하는 것이다. 대표적인 고차원 데이터는 사용자의 로그 데이터이다. 그림 2는 사용자의 로그 데이터로부터 k-NN그래프를 생성하는 것을 나타낸다. 사용자의 로그데이터란, 웹에서 사용자가 뉴스를 클릭한 로그, 사용자가 구매한 상품의 로그, 사용자가 본 영화의 로그 등 이진수의 값(binary weight)이 될 수 있다. 뉴스, 상품이나 영화는 그 종류가 굉장히 많기 때문에 고차원(high-dimension)의 데이터라고 할 수 있다. 다른 고차원의 데이터로는 bag-of-word형식의 텍스트 데이터와 이미지 데이터 등이 있다.

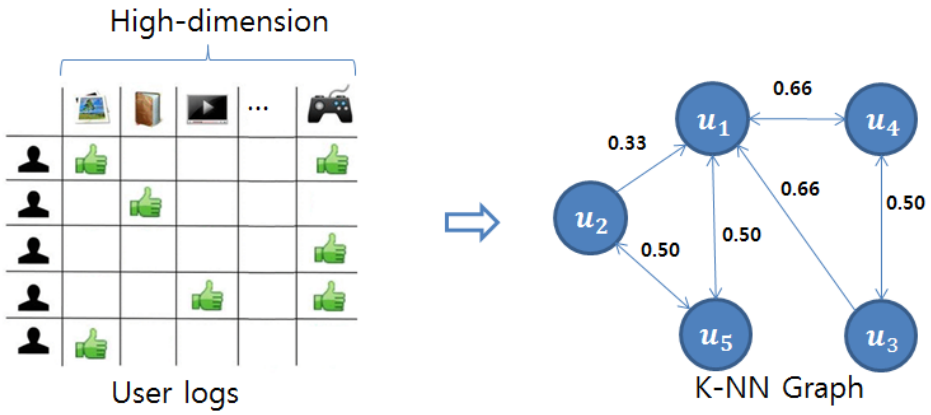


그림 2 사용자 로그 데이터로부터 k-NN그래프 생성

k-NN그래프는 웹과 관련된 많은 애플리케이션에서 중요한 연산이다. 사용자 기반 협업 필터링(user-based collaborative filtering)[1]에서 k-NN그래프는 비슷한 로그패턴을 가진 사용자를 연결하여 그래프를 생성하고, 그래프에서 사용자의 이웃(neighbor nodes)에 기반하여 추천한다. 이뿐만 아니라, 내용기반 검색 시스템(contents-based search system)에서는 데이터 셋의 변화가 없을 때, k-NN그래프 생성하는 방법이 k-NN탐색

을 하는 것보다 더 바람직한 방법이다[2]. 또한, k-NN그래프는 이상 값 탐지(outlier detection)[3], 이미지 분류(image classification)[4] 방법의 핵심 데이터구조이다. 이렇듯 여러 추천 시스템 및 정보검색에서 k-NN그래프를 다양하게 활용하고 있다. K-NN그래프는 이와 같이 여러 방법에 굉장히 중요한 역할을 하고 있어, 최근 들어 활발히 연구되고 있다[5].

정확한 k-NN그래프를 생성하는 가장 간단한(brute-force)방법은 모든 노드의 쌍에 대하여 유사도를 계산하고, 계산된 유사도 값을 기반으로 각 노드의 k-NN을 찾는 것이다. 즉, 노드가 n개 있을 때 각 노드로부터 나머지 (n-1)개의 노드와 유사도를 측정해야 하기 때문에 $O(n^2)$의 시간복잡도를 요구한다. 이는 대규모 문제에 대한 처리에서는 scalable하지 않아 시간이 너무 오래 걸리는 문제가 있다. 90%이상의 정확도를 가진 k-NN그래프를 활용하는 경우, 100% 정확도를 가진 k-NN그래프를 사용하는 경우와 성능 차이가 크지 않기 때문에[6], [7] k-NN그래프 생성 속도를 증가시키기 위해서 근사 k-NN그래프를 생성 알고리즘을 사용한다[2], [5].

이전의 NN탐색 문제에 대해서 K-D trees[8], R-tree[9] 등 트리기반의 인덱싱(tree-based indexing)방법이 사용되어 왔다. 하지만 이 방법들은 고차원(high-dimension)의 데이터에 대해서 효율적이지 않으며, 10이상의 대단히 높지 않은 차원의 데이터에서도 linear scan방법보다 비효율적이다[10]. 반면에, LSH(Locality Sensitive Hashing)방법은 고차원의 데이터에 대해서도 효율적이며[11], [12], 대규모(large-scale) 그룹 클러스터링에 적합한[13] 알고리즘이다, LSH의 핵심은 벡터간 유사성이 보존되도록 해시(hash)를 하여, 유사한 노드가 높은 확률로 같은 버켓 안에 들어가게 한다.

근사 k-NN그래프를 만드는 대표적인 방법은 LSH를 사용하여 먼저 유사한 노드를 그룹핑하고 그 그룹 안에서 brute-force한 방법으로 유사도를

계산하는 방법이다[5], [14]. 즉, 이웃후보 그룹을 생성하는 과정과 이웃후보 그룹에서 유사도를 계산하는 과정으로 이루어진다. 이러한 방법을 사용하면 유사도를 계산하는 횟수를 줄이면서 유사도가 높은 노드를 찾을 수 있다.

k-NN그래프 생성의 과정은 각각 한 개의 MapReduce 잡으로 구성되는데, 다음과 같은 이유에서이다. LSH를 단일 노드 시스템에서 구현하였을 때의 단점은 좋은 품질의 결과를 생성하기 위해서는 여러 개의 해시테이블이 필요하며, 이는 많은 개수의 인덱스를 메모리상에 유지해야 한다는 것이다. 이는 맵리듀스를 이용한 분산환경에서 인덱스를 메시지화 하기 때문에 해결할 수 있다. 또한, k-NN그래프 생성과정에서 LSH로 버켓에 매핑 한 후 같은 버켓에 속한 모든 쌍에 대해서 유사도를 검사하는데 상당한 시간을 소요하는 이 과정을 분산 처리하여 효율적이게 할 수 있다.

k-NN그래프 생성을 위한 두 과정 중 이웃후보 그룹을 생성하는 과정은 다음 두 가지가 중요하다. 유사도가 높은 사용자를 같은 그룹으로 만들어야 하며 각 그룹은 가능한 작아야 한다. 그룹의 크기가 커지게 되면 두 번 째 과정인 유사도를 계산하는 맵에서 시간이 오래 걸리기 때문이다. 뿐만 아니라 하둡 환경에서 큰 그룹의 경우 맵-사이드 스큐의 원인이 되는 비용이 큰 레코드(Expensive Record) [15]이기 때문에 맵-사이드 스큐가 발생할 수 있다.

본 논문에서는 맵리듀스 환경에서 LSH기반의 효율적인 k-NN그래프 생성 알고리즘에 대한 연구를 진행하였다. 기존의 k-NN그래프 탐색과 생성 알고리즘에서 발생하는 문제를 정리하였고 효율적인 이웃후보그룹생성을 위한 두 가지 알고리즘을 제안하였다.

논문의 구성은 다음과 같다. 2 장에서 근사 그래프 생성에 대한 기존 방법들을 살펴보고 3 장에서는 본 논문의 기본 도구가 되는 LSH기법과 아

파치 하둡에 대해 설명한다. 4 장에서는 본 논문의 k-NN그래프 생성 방법에 대해 자세히 설명한다. 5 장에서 실험 결과를 제시하고 6 장에서 결론 및 향후 연구에 대해 언급한다.

제 2 장

관련 연구

주어진 노드로부터 유사도를 구하고자 할 때, 한번에 모든 쌍의 유사도를 계산하는 것은 $O(n^2)$의 시간복잡도를 요구하기에 scalable하지 않다. 또한, 한 개의 해시테이블을 이용하여 사용자를 매핑하면, 적어도 한 개의 같은 아이템에 대한 평점을 공유하는 경우 같은 그룹에 들어가게 된다. 이는 빈도가 높은 아이템 때문에 일부 그룹의 크기가 커지는 문제가 있다. 위의 문제를 해결하고자 [1]에서는 맵리듀스 환경에서 LSH를 이용하여 그룹을 만들어주는 방법을 소개하였다. 해시함수는 MinHash[16]를 사용하였다. MinHash는 해시 충돌이 일어날 확률이 자카드 유사도와 같은 해시 함수이다. 이를 바탕으로 유사도가 높은 쌍을 찾아주기 위해서 LSH로 그룹을 만들어준 후 그룹내의 모든 쌍에 대해서 유사도 계산을 하는 방법이 소개되었다[5], [14], [17]. [14]에서는 MinHash를 이용하여 그룹을 만들어주고 그룹 내에서 유사도를 계산하여 k-NN그래프를 생성하고, [17]에서는 MinHash, LSH방법으로 일정한 유사도 이상의 쌍을 찾아주는 $\varepsilon - NN$탐색을 한다. 그룹내의 모든 쌍에 대해서 유사도를 계산하기 때문에 다음 두 가지 조건이 중요하다[5].

1. 유사도가 높은 노드는 같은 그룹에 속하여야 한다.
2. 각 그룹의 사이즈는 가능한 작아야 한다.

[14], [17]에서는 [1]에서의 그룹 사이즈를 작게 하는 방법을 사용한다. 이 방법은 한 개의 MinHash값으로 그룹을 생성하면 빈도가 높은 아이템 때문에 큰 그룹이 생성되는 것을 막고자 k개의 MinHash값을 이어 붙여서 그룹을 생성 한다. 그림 3의 경우 1개의 MinHash값으로 그룹을 생성 했을 경우와 2개의 MinHash값으로 그룹을 생성 했을 경우를 보여준다.

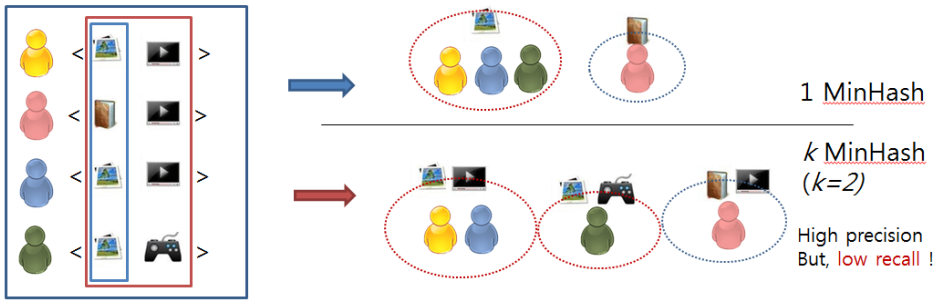


그림 3 MinHash를 사용한 그룹 생성의 예

만약 u_1과 u_2의 자카드 유사도가 20%라고 가정하면, k가 1일 때 해시 충돌이 일어나서 같은 그룹에 속할 확률도 0.2이 된다. 이때, k를 2로 증가시키면 두 개의 해시값이 같게 되어 같은 그룹에 속할 확률은 0.2^2이 된다. 즉, 유사도는 0에서 1의 값을 갖기 때문에 k를 증가시키면 같은 그룹에 속할 확률이 줄어들어 작은 그룹의 클러스터가 만들어진다. 하지만 이 방법은 큰 그룹뿐만 아니라 작은 그룹도 더 작게 만들어서 충분한 쌍을 찾지 못한다는 단점이 있다. 따라서, 위와 같은 n명의 사용자에 대한 그룹생성과정을 q번 수행하여 그림 4와 같이 좀 더 많은 쌍을 갖는 그룹들을 찾고자 하였다.

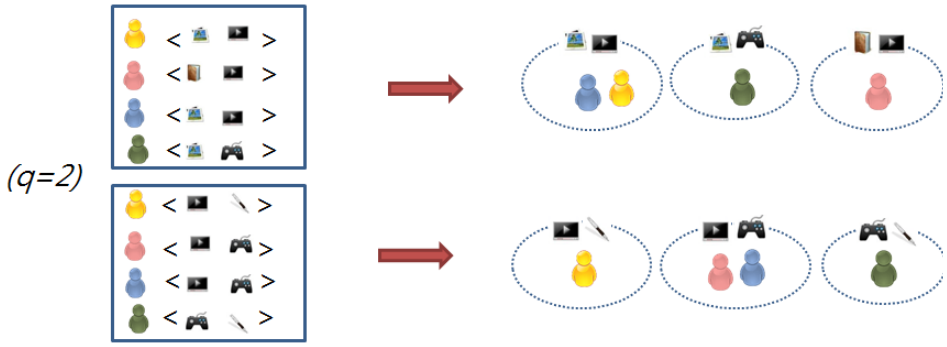


그림 4 q개의 다른 k MinHash를 사용한 그룹 생성의 예

q개의 다른 k MinHash를 이용하여 그룹을 만들 때, u_1과 u_2의 자카드 유사도가 s라고 가정하면 다음과 같은 속성을 갖는다.

A. u_1과 u_2가 한 그룹에 들어갈 확률: s^k

B. u_1과 u_2가 한 그룹에 들어가지 않을 확률: $1 - s^k$

C. u_1과 u_2가 q개의 그룹 중 한 그룹 이상에 들어갈 확률: $q * s^k$

D. u_1과 u_2가 q개의 그룹 중 어느 그룹에도 들어가지 않을 확률: $(1 - s^k)^q$

앞의 예와 같이, 만약 u_1과 u_2의 자카드 유사도가 20%라고 가정하면 표 1과 같다. k를 2로 증가 함으로써, k가 1이었을 때와 u_1과 u_2가 q개의 그룹 중 한 그룹 이상에 들어갈 확률을 비슷한 수준으로 맞추기 위해서는 q를 2에서 10으로 증가하여야 함을 보여준다.

	s^k	$q * s^k$	$(1 - s^k)^q$
k=1, q=2	0.2	0.4	0.64
k=2, q=5	0.04	0.2	0.327
k=2, q=10	0.04	0.4	0.66

표 1 q개의 다른 k MinHash에서 k의 변화에 따른 영향

이렇듯 이 방법은 이웃후보 생성과정에서 중간 생성 데이터가 늘어나게 되고 이웃후보 생성과정의 시간을 q배만큼 더 늘리기 때문에 k-NN그래프 생성을 함에 있어서 한계가 있다.

큰 그룹 사이즈를 방지하기 위한 다른 방법으로 [5]에서는 전체 노드에 대해서 해시의 값을 더하여 정렬한 후 일정한 크기의 블록 사이즈로 그룹을 나누어 모든 그룹의 사이즈를 같게 만드는 알고리즘을 소개하였다. 이 알고리즘은 빠르고 정확하며, 일반적인 유사도에서 포괄적으로 이용이 가능하다고 주장한다. 하지만 이 방법은 병렬처리 알고리즘이 아니며, 전체 노드에 대해서 해시의 값으로 정렬하기 때문에 메모리를 많이 차지한다는 단점과 해시의 값이 다른 노드들도 같은 그룹 내에 들어갈 수 있다는 단점이 있다. 위 방법에서는 128GB의 RAM으로 실험을 하였는데 우리의 실험환경에서는 2만개의 노드를 가진 데이터에 대해서 힙(heap)사이즈가 충분하지 않아서 전체 노드에 대해서 정렬을 하는 것이 불가능했다.

제 3 장

배경지식

이번 장에서는 본 논문의 기본 도구가 되는 LSH기법과 아파치 하둡에 대해 간략하게 설명한다.

3.1 LSH(Locality Sensitive Hashing)

LSH는 근사 k-NN 그래프 생성 문제에서 효율적인 기법으로 본 논문의 기초가 되므로 본 절에서 간략하게 소개한다. LSH는 그림 5와 같이 고차원(high-dimensional)의 데이터를 저 차원의 데이터로 바꾸는 것으로, LSH의 핵심은 벡터간 유사성이 보존되도록 해시(hash)를 하는 것이다.

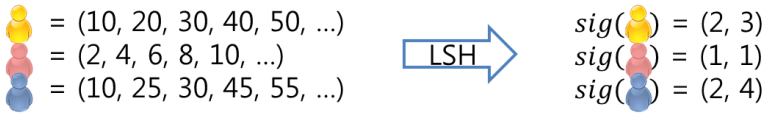


그림 5 LSH의 예

그러므로, 유사한 노드가 높은 확률로 같은 버켓(bucket)안에 들어가게

된다. 전반적으로, k-NN탐색에서 LSH는 2단계의 과정을 가진다. 처음 단계에서는 해시함수를 통해서 k개의 해시코드(hash code)를 만든다. 이 k개의 해시코드를 시그니처(signature)라고도 한다. 따라서, LSH는 각 노드를 h(x)로 해시 매핑(mapping) 함으로써 해시코드로 표현한다. 그리고 그 코드 값으로 각 노드를 버켓에 넣어 인덱싱(indexing)함으로써 해시테이블(hash table)을 만든다. 두 번째, 쿼리(query)단계에서는 쿼리를 해시코드로 바꾸고 쿼리의 해시코드로 인덱스 된 버켓 안의 근사 k-NN을 찾는다. LSH기법의 매력적인 특징은 이론적으로 최악의 경우에도 성능을 보장하며 [11], 실제로도 constant 하거나 sub-linear 탐색시간이 걸린다[5].

3.1.1 MinHash

MinHash[16]는 자카드 유사도를 같는 LSH이다. MinHash(Minwise 해싱)은 주어진 집합 W에 무작위로 순열을 바꾼 것(Random permutation) $\pi: \Omega \rightarrow \Omega$ 의 가장 작은 값을 저장하는 Random permutation방법을 대체한다. 다음과 같이 정의 한다.

$$h_{\pi}^{min}(\mathrm{W}) = \min(\pi(\mathrm{W}))$$

집합 W_1, W_2가 주어졌을 때, 다음을 따른다.

$$\Pr\left(h_{\pi}^{min}(W_1) = h_{\pi}^{min}(W_2)\right) = \frac{|W_1 \cap W_2|}{|W_1 \cup W_2|}$$

LSH에는 코사인 유사도(cosine similarity)를 측정하는 Random Projection[18]방법도 있는데, 최근 연구에 따르면 top-k nearest neighbor search를 할 때, real-valued vector 데이터의 경우 binary-valued로 이진화 한 후 MinHash를 사용하는 것이 이진화 된 데이터를 Random Projection방법으로 구하는 것보다 cosine similarity로 top-k

가 되어있는 것에 더 가까웠다. 이 뿐만 아니라 이진화 한 후 MinHash를 사용하는 것이 이진화 하지 않은 real-valued vector 데이터를 Random Projection방법으로 구하는 것 보다 더 좋은 결과를 얻었다[19]. 따라서 MinHash방법은 binary-valued vector 데이터뿐만 아니라 real-valued vector 데이터에도 유용하게 쓰일 수 있다.

3.2 아파치 하둡(Apache Hadoop)

아파치 하둡[20](Apache Hadoop, High-Availability Distributed Object-Oriented Platform)은 대량의 자료를 처리할 수 있는 큰 컴퓨터 클러스터에서 동작하는 분산 응용 프로그램을 지원하는 프리웨어 자바 소프트웨어 프레임워크이다. 분산처리 시스템인 구글 파일 시스템을 대체할 수 있는 하둡 분산 파일 시스템(HDFS: Hadoop Distributed File System)과 맵리듀스(MapReduce)[21]를 구현한 것이다.

3.2.1 맵리듀스(MapReduce)

맵리듀스[21]는 구글에서 대용량 데이터 처리를 분산 병렬 컴퓨팅에서 처리하기 위한 목적으로 제작하여 2004년에 발표한 소프트웨어 프레임워크다. 맵리듀스는 데이터 복제와 각각의 데이터 노드에서 지역적으로 계산을 수행하는 능력을 제공하는 분산파일시스템에 저장되어있는 데이터를 처리하는데 적합하다[1], [14], [17], [22].

맵리듀스 프레임워크는 이름에서 시사하듯, 맵(Map)과 리듀스(Reduce)라는 두 개의 기능적인 프로그래밍(functional programming)단계로 이루어져 있다. 계산의 입력(input)은 (key, value)의 쌍의 집합이며, 출력

(output) 또한 (key, value)의 쌍의 집합이다. 본 논문에서는 (key, value)의 쌍을 꺾쇠 괄호를 사용하여 <key, value>로 정의한다. key는 주로 리듀스 단계에서 어떤 value들이 서로 합쳐져야 하는지를 결정하는데 쓰인다. value는 임의적인 정보(arbitrary information)을 묘사한다. 두 과정은 아래와 같다.

$$\boldsymbol{Mapper}: < key1, value1 > \rightarrow \text{list} < key2, value2 >$$
$$\boldsymbol{Reducer}: < key2, \text{list} < value2 >> \rightarrow \text{list} < key3, value3 >$$

그림 6는 하둡 분산파일시스템에서의 맵리듀스 프레임워크를 보여준다. 계산은 분산파일시스템에 저장된 여러 개의 splits인 입력데이터를 병렬로 읽고 실행하는 맵 단계에서 시작한다. 각각의 split의 처리는 하나의 맵이 맡는다. 맵 단계에서 사용자가 정의한 맵 로직을 수행하고, 그 결과를 <key, value> 형태로 변환하여 출력한다. 각 맵 단계의 출력은 중간키 값으로 해시 파티션(hash-partitioned)된다. 리듀스 전 단계에서 각 파티션은 key값으로 정렬되고 합쳐지는 shuffle단계를 거쳐 같은 key를 갖는 모든 파티션은 하나의 리듀스로 보내진다. 맵 단계에서 맵 함수가 한번에 수행하는 단위가 입력 텍스트의 한 줄 단위였다면, 리듀스 함수가 수행하는 단위는 한 개의 정렬된 key에 대한 value의 리스트 형태이다. 따라서 Reducer의 입력 형태 는 <key2, list<value2>> 형태가 된다. 그리고 Reducer에서는 Mapper에서와 마찬가지로 사용자가 정의한 리듀스 함수를 수행하여 최종결과를 만든다. 각 리듀스 함수의 결과를 DFS에 저장하고 맵리듀스 과정이 종료된다.

하둡 분산 파일 시스템(HDFS, Hadoop Distributed File System)은 하둡 프레임워크를 위해 자바언어로 작성된 분산 확장 파일시스템이다. HDFS은 여러 기계에 대용량 파일들을 나눠서 저장을 한다. 데이터들을 여

러 서버에 중복해서 저장을 함으로써 데이터 안정성을 얻는다.

본 논문에서는 k-NN그래프 생성을 위하여 오픈소스인 하둡 맵리듀스 프레임워크를 사용하였다. 맵리듀스는 분산 병렬 시스템에서 대용량 데이터를 처리하기 위해 고안된 프레임워크로써 그림 6 처럼 데이터를 특정 함수 및 패턴에 따라 키와 값 집합으로 나누는 맵 단계와 이 집합을 키 별로 합치는 리듀스 단계로 구성된다. 본 논문에서는 총 두 번의 맵리듀스 과정을 수행한다. 하둡 분산 파일 시스템 (HDFS)은 대용량 데이터를 여러 노드에 분산시켜 병렬적인 맵 연산 작업을 처리한다.

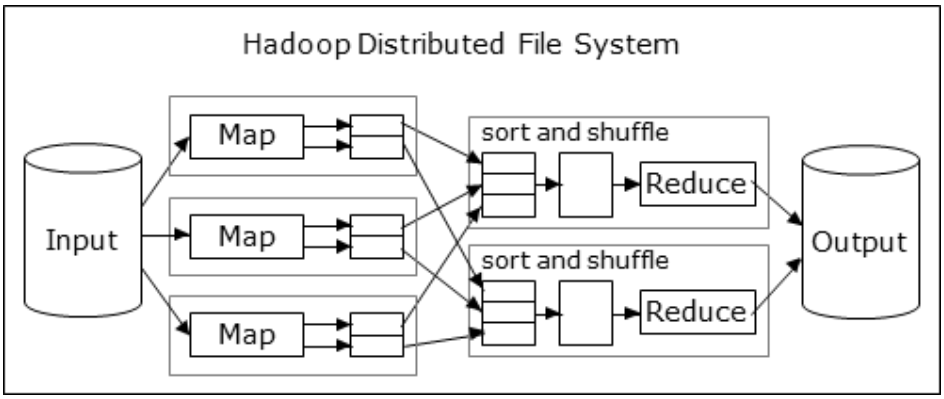


그림 6 하둡 분산파일시스템에서의 맵리듀스 프레임워크

제 4 장

LSH를 이용한 k-NN 그래프 생성

본 장에서는 본 논문의 k-NN그래프 생성 방법에 대해 자세히 설명한다.

4.1 문제 정의

n개의 노드의 집합 $U = \{u_1, u_2, \ldots, u_n\}$과 유사도 $S(u_i, u_j)$가 주어졌을 때, U의 k-NN그래프는 u_j가 u_i의 가장 유사한 k개의 노드에 속할 때, 노드 i에서 j로 간선이 있는 방향그래프(directed graph)이다. 본 논문에서 실험한 사용자로그 데이터의 경우 사용자가 노드이며, 각 노드는 아이템 평점을 준 로그 셋L_u 을 갖는다. S는 자카드 계수(Jaccard coefficient)를 사용하였고 다음과 같이 표현 할 수 있다.

$$S(u_i, u_j) = \frac{\left| L_{u_i} \cap L_{u_i} \right|}{\left| L_{u_i} \cup L_{u_i} \right|}$$

즉, 각 사용자와의 자카드 계수가 높은 k개의 사용자를 찾아주는 그래프를 만드는 것이다. 텍스트 데이터의 경우에는 각 문서와의 자카드 계수가 높은 k개의 문서를 찾아주는 그래프를 만들게 된다. k-NN그래프를 생성하기 위해서 그림 7와 같이 크게 이웃후보 그룹을 생성하는 과정과 이웃후보

그룹 내에서의 유사도를 계산하는 과정을 갖는다. 각 과정은 한 개의 맵-리듀스로 구현되어 총 2개의 맵-리듀스 잡을 통하여 k-NN그래프를 생성한다.

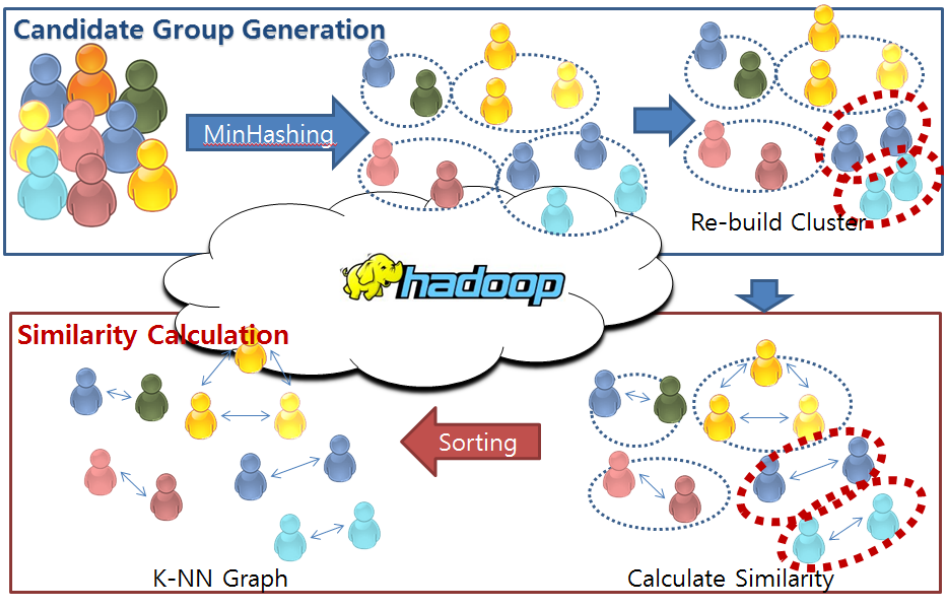


그림 7 k-NN 그래프 생성을 위한 과정

4.2 MinHash를 이용한 이웃후보 그룹생성

본 절에서는 MinHash를 이용한 이웃후보 그룹을 생성하는 것에 대하여 설명한다.

MinHash[16]는 LSH기법의 한가지로 자카드 계수(Jaccard coefficient)를 유사도로 갖는다. 유니버셜 해시함수를 사용하며, [1]에서는 한 사용자에 대해서 1개의 해시함수를 사용하여 작은 k개의 해시값을 사용자의 그룹ID로 만들어 그룹을 만든다. 그리고 이 작업을 q번 반복하여 한 사용자에 대해서 q개의 그룹ID를 만든다. 그리고 이 작업을 seed가 다른 해시함수로 q

번 반복 수행하여 한 사용자에 대해서 q개의 그룹ID를 만든다. 본 논문에서도 위와 같은 방법을 사용하여 첫 번째 맵에서 알고리즘1과 같이 그룹ID를 만들어 유사도 후보그룹을 만들었다. MinHash의 정의역은 한 사용자의 아이템 집합이며, 치역은 64bit의 long value이다. q개의 그룹ID를 만들기 위해서 MinHash에 사용되는 seed를 q개 사용하였다. 즉, 한 사용자는 q개의 다른 그룹ID가 생성되고 이는 한 사용자는 q개의 그룹에 속함을 의미한다.

알고리즘 1. 후보 그룹 생성: 맵
Algorithm 1. Candidate group generation Map

```
Input file: user, list<item>
Map(k,q)
read input file
for each user do
    for i..q do
        Hashed vector = MinHash(list<item>)
        group-id = 1~k smallest values in Hashed vector
        Emit(group-id, user)
    end for
end for
```

4.3 이웃후보 그룹 재구성

본 절에서는 첫 번째 리듀스에서 사이즈가 큰 그룹에 대해서 처리해주는 두 가지 방법을 제시한다.

4.3.1 일정한 그룹사이즈로 그룹 재구성

첫 번째 방법은 사이즈가 큰 그룹을 일정한 사이즈로 나누어 그룹을 재구성 하는 방법이다. 이 방법은 [5]에서 최대 사이즈를 제한한 방법이 모티브가 되었다. [5]의 알고리즘은 맵리듀스 환경의 알고리즘은 아니지만 최대 사이즈를 제한함으로써 사이즈가 큰 그룹을 생성하지 않는다. 전체 노드에 대해서 해시의 값으로 정렬한 후 일정한 크기의 블록 사이즈로 그룹을 한다. 전체 노드에 대해서 정렬을 하기 때문에 메모리를 많이 차지한다. 위 방법에서 128GB의 RAM으로 실험을 하였는데 본 논문의 실험환경에서는 6만개의 노드를 가진 데이터에 대해서는 힙(heap) 사이즈가 충분하지 않아서 정렬을 하는 것이 불가능했다. 또 다른 단점은 해시의 값이 다른 노드들도 같은 그룹 내에 들어갈 수 있다는 점이다. 본 논문의 첫 번째 방법에서는 알고리즘 2와 같이 해시값이 같은 노드만 그룹에 들어가게끔 먼저 그룹을 만들어준 후 일정블록 사이즈가 넘는 그룹의 경우 그룹 아이디를 바꿔서 그룹을 일정한 블록사이즈의 그룹으로 나누는 방법이다. 이 방법은 기존의 방법에 추가적인 데이터 구조를 만들지 않아도 되며 추가적인 계산을 하지 않기 때문에 추가비용이 적다.

알고리즘 2(a). 후보 그룹 생성: 리듀스

Algorithm 2(a). Candidate group generation Reduce

```
Input: group-id, list<user>, group-size threshold
Reduce(input)
for user in list<user> do
    userList.add(user)
    if userList.size > group-size threshold
      Emit(group-id + i++, userList)
      userList.clear
    Emit(group-id, userList)
end for
```

그림 8에서는 임계 블록사이즈 값이 3인 예를 보여주고 있다. 사용자에 대해서 1개의 해시 값을 만들고 그 해시 값으로 그룹을 구성한다. 그리고 임계 블록사이즈 값을 넘는 그룹에 대해서 그룹사이즈를 3이하로 재구성한다.

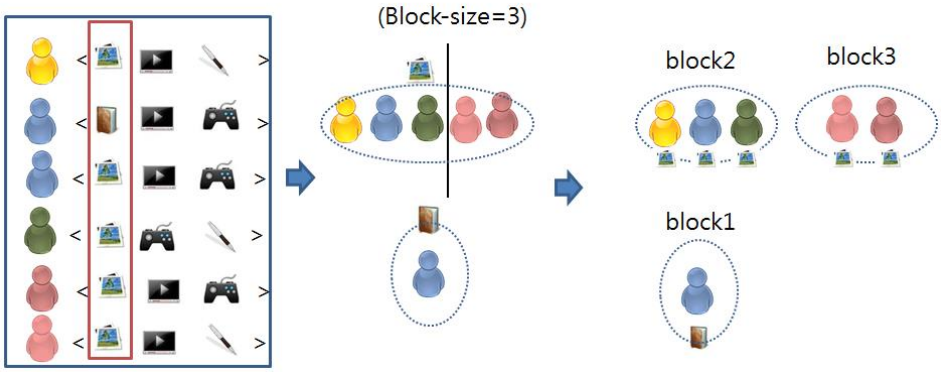


그림 8 일정한 그룹사이즈를 초과하지 않는 그룹 재구성

다른 예를 들면, A그룹에 사용자 120명이 속해있고, 임계 블록사이즈

가 50일때, A그룹은 A_0그룹50명, A_1그룹50명, A_2그룹20명으로 나누어진다. 위와 같은 방법으로 그림 9과 같이 그룹을 나누면 다음 과정인 유사도 계산 과정에서 유사도 계산을 하는 쌍을 줄이게 되는 장점을 이용하였다. 막대그래프에서 y축은 그룹의 사이즈, x축은 그룹을 나타낸다.

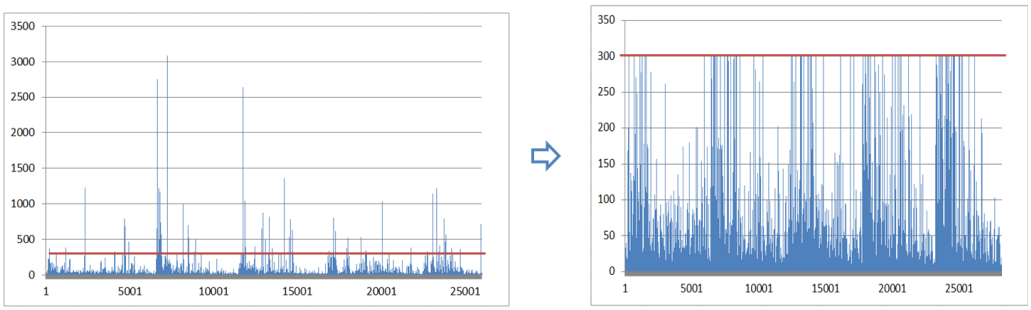


그림 9 그룹 재구성의 결과

전체사용자를 n, 임계 블록사이즈를 $blockSize$라고 할 때, 한 그룹에 대해서 최악의 경우 유사도 계산의 시간복잡도를 $O(n^2)$ 에서 $O(blockSize^2)$로 줄였으며, 전체 사용자에 대하여 MinHash테이블을 만드는 과정을 q번 반복 할 때, 두 번째 과정에서 한 사용자에 대하여 가장 유사도가 높은 사용자를 추출하는 과정에서의 정렬에 대한 최악의 경우 시간복잡도를 $O(n * log\, n)$ 에서 $O(q * blockSize \log(q * blockSize))$ 로 줄였으며 공간복잡도를 $O(n)$에서 $O(q * blockSize)$로 줄여서 메모리에 부담을 덜어주었다.

4.3.2 재 해싱을 통한 그룹 재구성

두 번째 방법은 재 해싱(re-hashing)을 통한 hierarchical LSH를 이용하여 사이즈가 큰 그룹을 작은 여러 개의 그룹으로 나누는 방법이다. 이 방법은 [23]에서 제시한 재 해싱 개념이 모티브가 되었다. [23]에서는 ε-NN 탐색을 할 때 쿼리포인트가 속한 그룹이 임계 값을 넘으면 그 그룹에 속한 모든 포인트에 대해서 재 해시를 하여 그림 10와 같이 그룹을 나누었다.

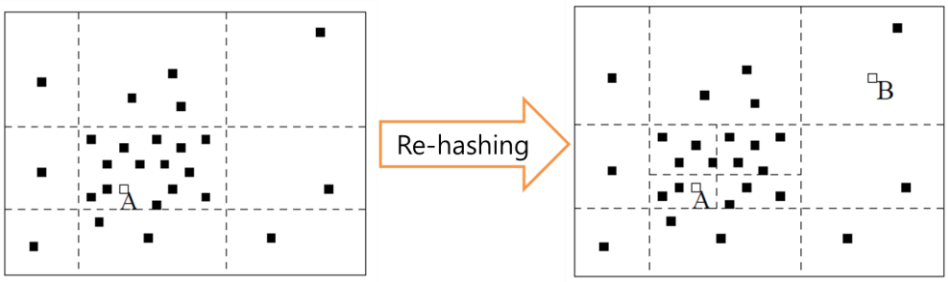


그림 10 재해싱 방법의 예

두 번째 방법 또한 첫 번째 방법과 마찬가지로 일정한 임계 값을 정하고 임계 값을 넘은 그룹의 경우 해싱을 한번 더 하여 해시 값을 하나 더 만들어서 그룹ID에 붙인다. 즉, 이 그룹의 경우 처음 해시를 했던 k값보다 하나 더 많은 k+1개의 해시 값을 그룹ID로 갖는다. 이렇게 하여 알고리즘 2(b)와 같이 그룹을 재구성한다. 만약 재구성한 그룹이 다시 임계 값을 넘는다면 재귀적으로 앞에서 설명한 방법으로 그룹을 재구성한다. 즉, 재구성한 그룹에 속한 사용자에 대하여 한번 더 해싱을 하여 k+2개의 해시 값을 그룹ID로 만든다. 따라서, 첫 번째 방법과 마찬가지로 모든 그룹의 사이즈는 임계 값을 넘지 않아 큰 그룹을 만들지 않는다.

알고리즘 2(b). 재 해시 후보그룹 생성: 리듀스

Algorithm 2(b). Candidate group generation Reduce

```
Input: group-id, list<user>, group-size threshold, k
Reduce(input)
for user in list<user> do
    userList.add(user)
    if userList.size > group-size threshold
        oversize=true
end for
    if oversize
      reBuild(userList, ++k, threshold)
    else Emit(group-id, userList)
reBuild(userList,k,threshold)
    map=makeKPlusMap(userList, k)
    for list in lists=listFromMap(map)
        if list.size<threshold
            Emit(group-id, list)
        else reBuild(list, ++k, threshold)
End reBuild
makeKPlusMap(list, k)
For user in list do
    newGroup-id =MinHash(user.items, k)
    map.add(newGroup-id, user)
Return map
```

첫 번째 방법과 다른 점은 재 그룹을 할 때 재 해시를 하여 kMinHash에서 k를 증가 함으로써 그림 11과 같이 더 유사도가 높은 쌍을 같은 그룹에 속하게 한다는 점이다. 첫 번째 방법은 모든 사용자의 경우 일정한 k값으로 그룹ID를 만들지만 두 번째 방법은 큰 그룹에 속한 사용자의 경우 k값이 더 큰 값으로 그룹ID를 만들게 된다. 따라서, 기존의 방법에서 k를 증가 시킬 때, 작은 그룹도 더 작아지는 문제가 있었는데 두 번째 방법을 사용하면 작은 그룹은 k가 더 큰 것으로 해싱하지 않기에 그대로 보

존 된다는 장점이 있다. 하지만 이 방법은 추가적으로 해시를 해야 하기에 추가시간이 들어간다. 전체사용자를 n, 임계블록사이즈를 $blockSize$라고 할 때, 재 해싱을 하지 않는 경우 Minhash를 하여 그룹을 만들어주는데 $O(n*k*d)$시간 복잡도가 걸렸는데 재 해싱을 하게 되면 $O(\varepsilon*k*d)$만큼의 시간복잡도가 추가되어 $O((n+\varepsilon)*k*d)$로 증가한다. ε는 n에 비하여 작은 수 이고 재 해싱 방법의 초기 k값을 1로 시작하지 않고 적절하게 올린 후 사용하기 때문에 추가되는 시간은 크지 않으며, k는 비교적 작은 수이기에 재 해싱을 하는 방법과 하지 않는 방법 모두 $O(n*d)$가 된다. 이 두 번째 방법으로 그룹을 나누면 다음 과정인 유사도 계산 과정에서 유사도 계산을 하는 쌍을 줄이게 되는 장점을 이용하였다. 한 그룹에 대해서 최악의 경우 유사도 계산의 시간복잡도를 $O(n^2)$ 에서 $O(blockSize^2)$로 줄였으며, 전체 사용자에 대하여 MinHash테이블을 만드는 과정을 q번 반복 할 때, 두 번째 과정에서 한 사용자에 대하여 가장 유사도가 높은 사용자를 추출하는 과정에서의 정렬에 대한 최악의 경우 시간복잡도를 $O(n*log\,n)$에서 $O(q*blockSize\log(q*blockSize))$로 줄였으며 공간복잡도를 $O(n)$에서 $O(q*blockSize)$로 줄여서 메모리에 부담을 덜어주었다.

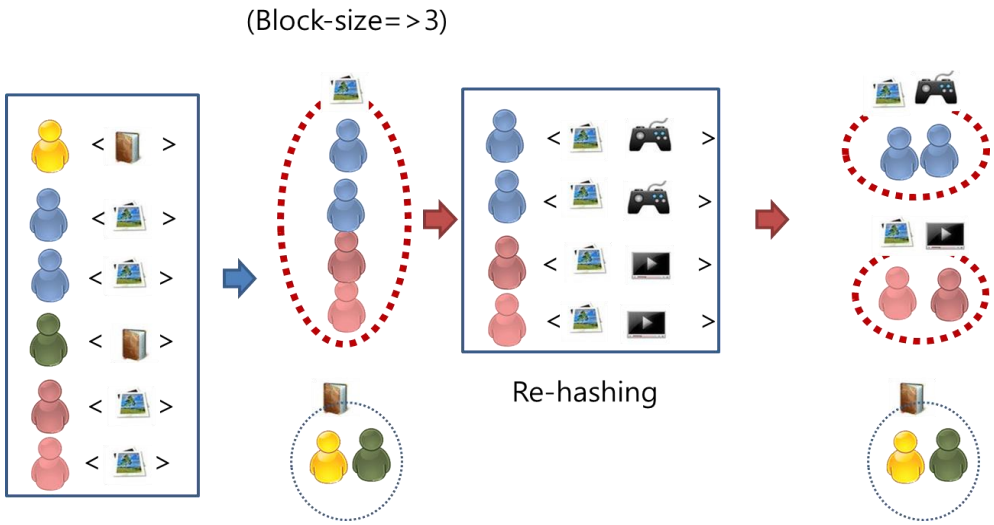


그림 11 재해싱을 통한 그룹 재구성

4.4 이웃후보 그룹내의 유사도 검사 및 k-NN 추출

이웃후보그룹이 완성된 후 이웃후보 그룹내의 모든 쌍에 대해서 유사도를 검사한다. 이 과정은 한 개의 맵-리듀스로 구성되는데 먼저, 맵은 알고리즘3과 같이 그룹내의 $\frac{\text{group size}(\text{group size}-1)}{2}$쌍을 만들고 각각의 쌍에 대해서 유사도를 계산한다. 그리고 아웃풋으로 사용자를 key로 후보사용자와 유사도값을 value로 하여 리듀스 단계로 보낸다. 리듀스에서는 각 사용자에 대하여 후보사용자와 유사도값의 리스트를 받게 되고 각 사용자에 대해서 후보사용자를 유사도 값으로 정렬하여 유사도가 높은 k명의 사용자를 찾는다. 앞에서 이웃후보 그룹 생성과정에서 그룹사이즈의 제한을 두었기에 이 과정에서 한 사용자에 대하여 가장 유사도가 높은 사용자를 추출하는 과정에서의 정렬에 대한 최악의 경우 시간복잡도를 $O(n * log\, n)$에서 $O(q * blockSize \log(q * blockSize))$ 로 줄였으며 공간복잡도를 $O(n)$ 에서 $O(q *$

blockSize)로 줄여서 메모리에 부담을 덜어주었다.

알고리즘 3. 유사도 계산

Algorithm 3. Similarity Calculation

```
Input: group-id, list of users
Map(intput)
make all pair   for users
for each user do
   sim=similarity calculation(ui, uj)
   Emit(ui, (uj,sim))
   Emit(uj, (ui,sim))
end for

Input: user, list<(user,sim)>   as key, values
Reduce(intput)
for each (user ,sim) in values do
    map.add(user,sim)
end for
Map.sort by sim
Emit(top k user,sim)
```

제 5 장

성능평가

본 실험은 잡트래커(job tracker)인 마스터(master) 컴퓨터 1대와 태스크트래커(task tracker)인 슬레이브(slave) 컴퓨터 10대의 하둡 클러스터에서 수행했다. 각 노드에 해당하는 컴퓨터는 3.1GHZ 쿼드코어 Intel Core i5-2400 CPU, 4GB RAM과 4TB의 하드디스크로 구성된다. HDFS의 블록 사이즈는 기본 설정인 64MB로 설정하였고, 각 태스크트래커는 동시에 3개의 맵 태스크와 3개의 리듀스 태스크를 할당 받을 수 있도록 설정하였다. 하지만 전체 태스크트래커는 동시에 최대 30개의 맵 태스크와 28개의 리듀스 태스크까지 수행할 수 있다.

5.1 실험설정

본 절에서는 본 논문에서 사용한 데이터 셋, 비교 알고리즘, 성능평가 기준에 대해서 설명한다.

5.1.1 데이터 셋

데이터는 무비렌즈(MovieLens)[24]에서 제공하는 무비렌즈 데이터와

뉴욕 타임즈 데이터[25]를 사용하였다. 무비렌즈 데이터 셋은 온라인 영화 추천 서비스인 무비렌즈의 사용자의 영화에 대한 1에서 5까지의 평점 (rating) 로그이다. 데이터 셋의 사용자는 적어도 20개의 로그가 있는 사용자 중에서 무작위로 선정되다. 본 논문에서는 이 데이터 셋을 [1]에서 사용한 방법으로 이진화 하였다. 각 사용자에 대해서 점수가 사용자의 평균점수보다 높을 경우 1로 하였고 그렇지 않을 경우 0으로 이진화 하였다.

무비렌즈 데이터는 표 2와 같이 71,567명의 사용자로부터 10,681개의 영화에 대한 10,000,054개의 평점 로그를 가지고 있다. 뉴욕 타임즈 데이터는 bags-of-word형식의 데이터이다. 기사에 대해서 Stop words를 제거하고 10번이상 나타나는 단어에 대하여 빈도수를 가지고 있다. 본 논문에서는 이 데이터를 이진화 하기 위해 [17]에서의 방법처럼 빈도가 있는 경우 1로 그렇지 않을 경우 0으로 이진화 하였다. 뉴욕 타임즈 기사에서 300,000개의 문서에 대하여 총 102,660의 단어가 나타나고 있다. 본 논문에서는 20,000개의 문서에 대하여 실험하였다.

	무비렌즈	뉴욕 타임즈
Size	71,567	20,000
Dimensionality	10,681	102,660

표 2 데이터 셋 설명

5.1.2 비교 알고리즘

본 실험에서 제시한 알고리즘과 기존의 알고리즘을 비교하기 위해서 본 실험에서 제시한 첫 번째 방법인 일정한 그룹사이즈로 재구성한 방법을 LSH-R로, 두 번째 방법인 재 해싱(re-hashing)을 이용한 K+방법을 LSH-

K+로 표현하고 [14]에서 제시한 방법의 그래프 생성 알고리즘을 LSH-B로 표현한다.

5.1.3 성능평가 기준

본 실험의 평가 기준으로는 Brute-force 방법과의 상대적인 유사도 계산 횟수 비율을 나타내는 Scan Rate와 정확도를 나타내는 Accuracy를 사용하였다. G가 정확한 k-NN그래프를 나타내고 $E(\cdot)$가 그래프에서의 간선(directed edge)를 나타내고 $|\cdot|$가 집합의 개수일 때, 근사 k-NN그래프 G'의 정확도는 다음과 같이 정의한다. 즉, 근사 k-NN그래프가 정확한 k-NN그래프와 많은 간선을 공통으로 가지고 있으면 정확도가 올라간다.

$$\text{Accuracy}(G') = \frac{|E(G') \cap E(G)|}{|E(G)|}$$

데이터의 전체 노드가 n개 있을 때, Brute-force방법은 $n(n-1)/2$번의 유사도 계산한다. 따라서, 정확한 k-NN그래프를 생성하기 위해서는 $n(n-1)/2$번의 유사도 계산이 필요하다. Scan Rate는 실제로 유사도를 계산한 횟수와 Brute-force방법에서의 유사도 계산 횟수의 상대적 비율이며 다음과 같이 정의한다. Scan Rate가 1이면 brute-force방법과 같은 횟수의 유사도 계산을 했다는 것이고 Scan Rate가 낮을수록 brute-force방법과 비교하여 적은 유사도 계산을 했다는 것을 나타낸다.

$$\text{Scan Rate} = \frac{\#\text{ actual similarity calculation}}{n(n-1)/2}$$

5.2 실험 결과

5.2.1 시간과 정확도

먼저, 기존 알고리즘의 한계를 보여주는 실험으로 뉴욕타임즈 데이터에 대하여 k를 다르게 하여 각각의 경우를 다양한 정확도의 그래프로 만들기 위해서 변수 q를 다르게 하여 실험하였다. 그림 12은 뉴욕타임즈 데이터에 대한 LSH-B알고리즘의 k변화에 따른 시간과 정확도 비교이다. K를 1로 하였을 때보다 k를 2로 하였을 때 더 짧은 시간에 더 정확도가 높은 그래프를 생성함을 볼 수 있다. 이는 기존의 방법의 효과를 본 것이다. 사이즈가 큰 그룹을 줄이게 되어 두 번째 잡에서의 병목을 없앴다. 하지만 k를 3으로 올리게 되면 k가 2일때보다 성능이 안 좋아 지는 것을 볼 수 있다. 2장에서 언급했듯이 변수 q가 커지게 되는 문제가 있음을 확인하였다.

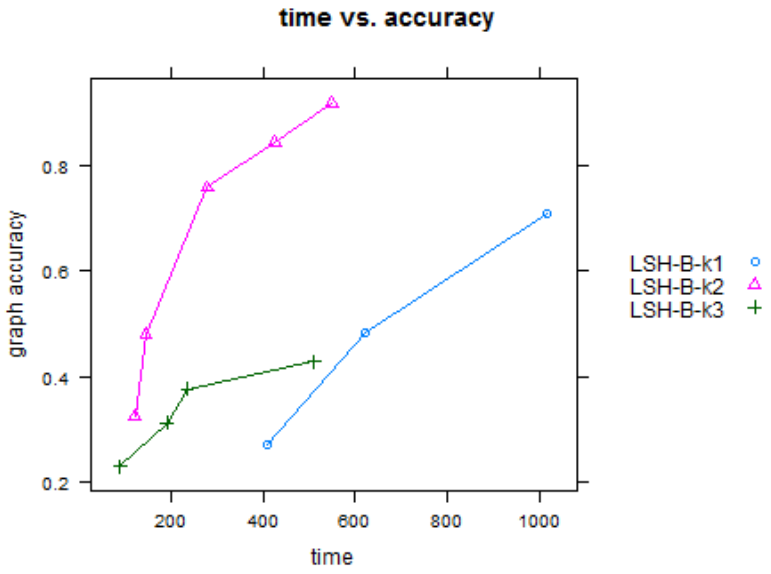


그림 12 뉴욕타임즈 데이터에서 LSH-B알고리즘의 k변화에 따른 시간과 정확도 비교

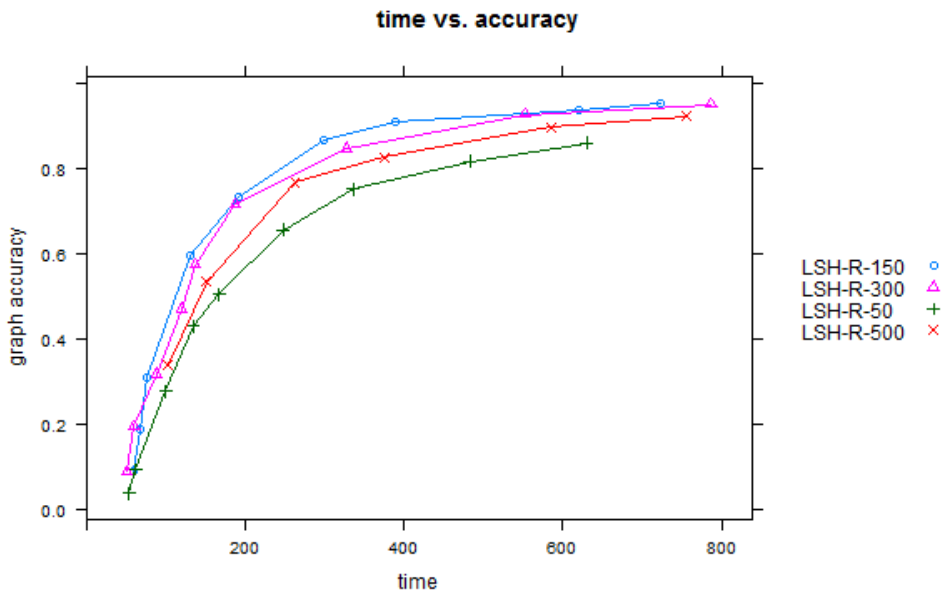


그림 13 무비렌즈 데이터에서 LSH-R알고리즘의 여러 최대 블록 사이즈에 대한 시간과 정확도 비교

본 논문에서는 최대 블록사이즈를 정하여 그룹을 재구성 하였다. 그림 13과 같이 블록 사이즈의 영향을 알기 위해서 블록사이즈를 다르게 하여 실험을 하였다. 블록사이즈는 k-NN에서 k보다 커야 하며, 물론 전체 노드의 개수 보다 작아야 한다. 블록사이즈=$\{50, 150, 300, 500\}$에 대하여 실험하였다. 블록사이즈가$\{150, 300\}$일 때가 블록사이즈가 $\{50, 500\}$일 때보다 성능이 좋았다. 블록사이즈가 50일때는 그래프를 생성하기 위해서는 너무 작고 500은 너무 컸기 때문이다. 그래서 실험에서 150으로 고정하여 실험하였다.

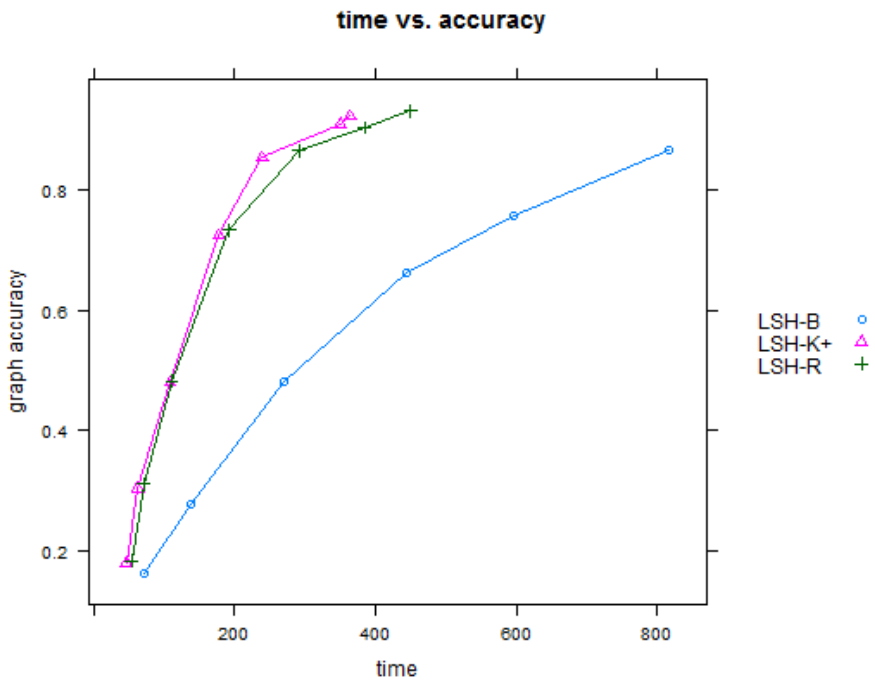


그림 14 무비렌즈 데이터에서 여러 알고리즘에 대한 시간과 정확도 비교

다양한 정확도의 그래프를 만들기 위해서 해시테이블을 만드는 개수(변수 q)를 다르게 하며 실험하였다. 그림 14는 무비렌즈 데이터에서 여러 가지 방법에 대해서 다양한 정확도의 그래프를 생성하는데 걸리는 시간을 측정하여 비교하였다. LSH-K+방법이 가장 빠른 시간에 90%정확도의 근사 그래프를 생성함을 확인하였으며, 이는 LSH-B방법보다 두 배 이상 빠르게 그래프를 생성함을 볼 수 있다.

그림 15은 뉴욕타임즈 데이터에서 여러 가지 방법에 대해서 다양한 정확도의 그래프를 생성하는데 걸리는 시간을 측정하여 비교하였다. 무비렌즈에서와 마찬가지로 LSH-K+알고리즘이 가장 빠른 시간에 90%이상의 정확도를 갖는 근사 그래프를 생성함을 확인하였으며, 이는 LSH-B방법보다

1.6배의 시간효율을 보였으며 LSH-R보다 10% 더 좋은 시간 효율을 보였다.

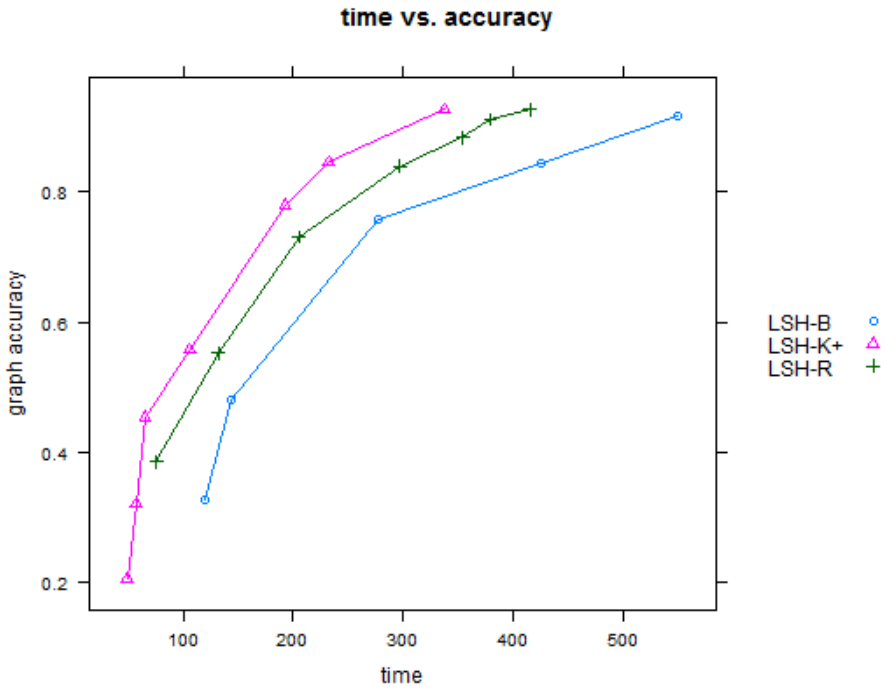


그림 15 뉴욕타임즈 데이터에서 여러 알고리즘에 대한 시간과 정확도 비교

5.2.2 정확도와 상대 유사도 계산비율(Scan Rate)

k-NN그래프 생성에 있어서 유사도를 계산하는 시간이 지배적이기 때문에 위에서 정의한 Scan Rate를 측정하여 비교하였다. 그림 16은 알고리즘이 brute-force한 방법과 비교하여 얼만큼의 유사도 계산을 하고 어떠한 정확도의 그래프를 생성하는지 나타낸다. x축은 Scan Rate를 나타내며, y축은 생성한 그래프의 정확도를 나타낸다. LSH-K+알고리즘이 가장 적은 유

사도 계산을 하면서 높은 정확도의 그래프를 생성한다. LSH-K+알고리즘의 경우 brute-force한 방법의 7.3%의 유사도 계산만으로 정확한 k-NN그래프와의 정확도 90%가 되는 k-NN그래프를 생성 할 수 있다. LSH-R알고리즘의 경우 정확한 k-NN그래프와의 정확도 90%가 되는 k-NN그래프를 생성하는데 brute-force한 방법의 8.8%의 유사도 계산이 필요했으며, LSH-B알고리즘의 경우 13% 이상의 유사도 계산이 필요했다.

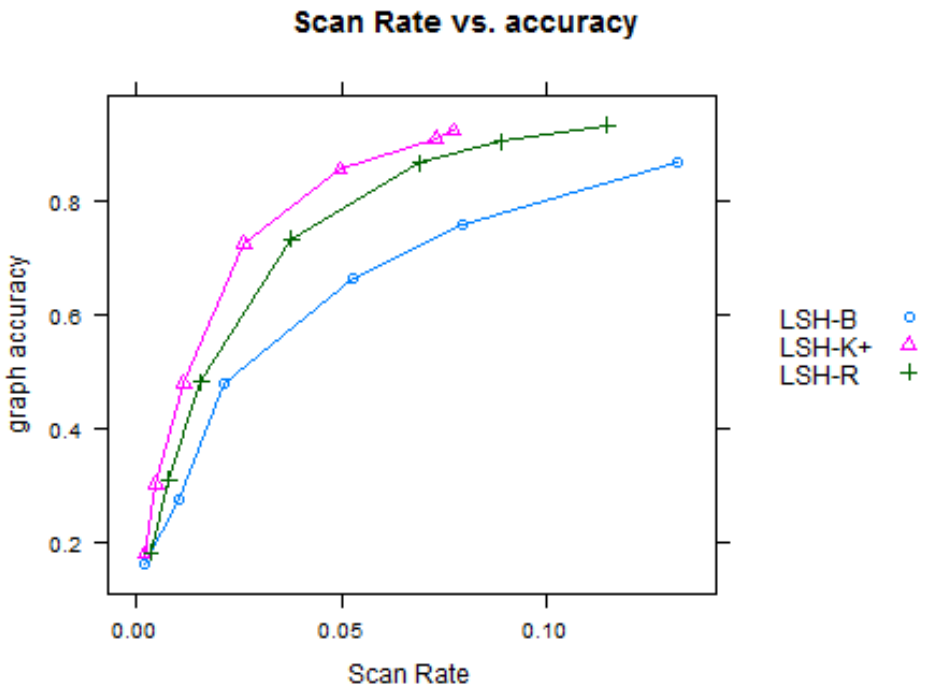


그림 16 무비렌즈 데이터에서 여러 알고리즘에 대한 Scan Rate와 정확도 비교

그림 17은 뉴욕타임즈 데이터에서 여러 가지 방법에 대해서 brute-force한 방법과 비교하여 얼만큼의 유사도 계산을 하고 어떠한 정확도의 그래프를 생성하는지를 비교하였다. 무비렌즈에서와 마찬가지로 LSH-K+알

고리즘이 가장 적은 유사도 계산을 하면서 높은 정확도의 그래프를 생성한다. LSH-K+알고리즘의 경우 brute-force한 방법의 23%의 유사도 계산만으로 정확한 k-NN그래프와의 정확도 90%가 되는 k-NN그래프를 생성 할 수 있다. LSH-R알고리즘의 경우 정확한 k-NN그래프와의 정확도 90%가 되는 k-NN그래프를 생성하는데 brute-force한 방법의 54%의 유사도 계산이 필요했으며, LSH-B알고리즘의 경우 31%이상의 유사도 계산이 필요했다.

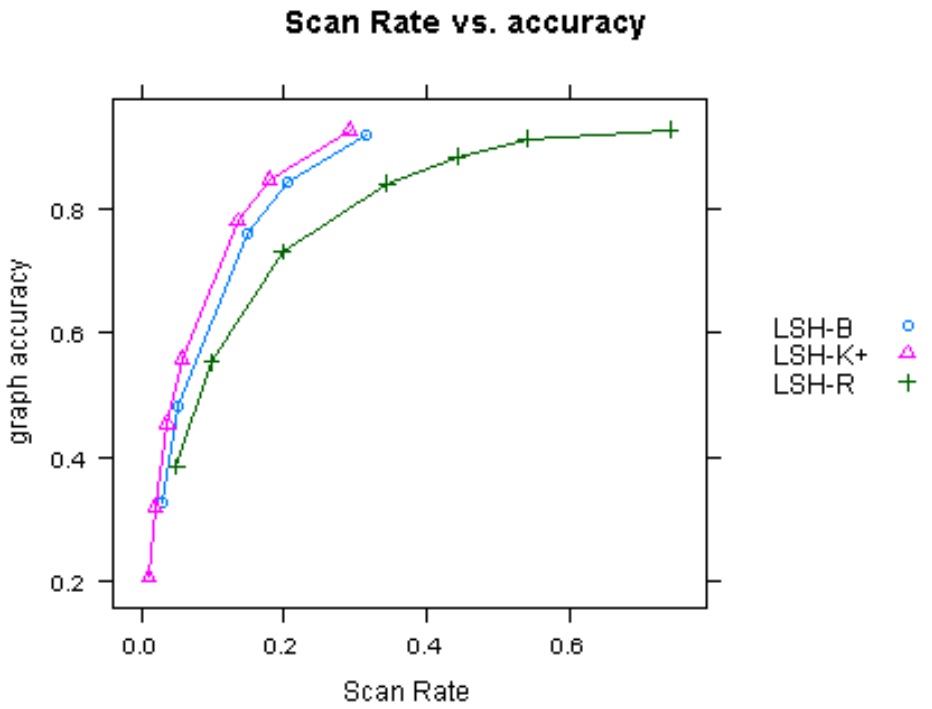


그림 17 뉴욕 타임즈 데이터에서 여러 알고리즘에 대한 Scan Rate와 정확도 비교

5.2.3 정확도와 해시테이블 개수

본 논문에서는 그래프 생성의 속도와 정확도를 조절하는데 전체사용자

에 대한 MinHash값을 만들어 주는 과정(이웃후보 생성과정)의 횟수(변수 q)를 사용하였다. 실제로 해시 테이블을 유지한 것은 아니지만 전체과정의 반복이기에 해시테이블이라고 표현하였다. 그림 18는 해시테이블과 정확도의 그래프이다. LSH-R알고리즘이 가장 적은 이웃후보 생성과정을 통해서 정확도가 높은 그래프를 생성 하였다. LSH-B의 경우 2장에서 언급했듯이 변수 q가 커지게 되는 문제가 있음을 확인하였다.

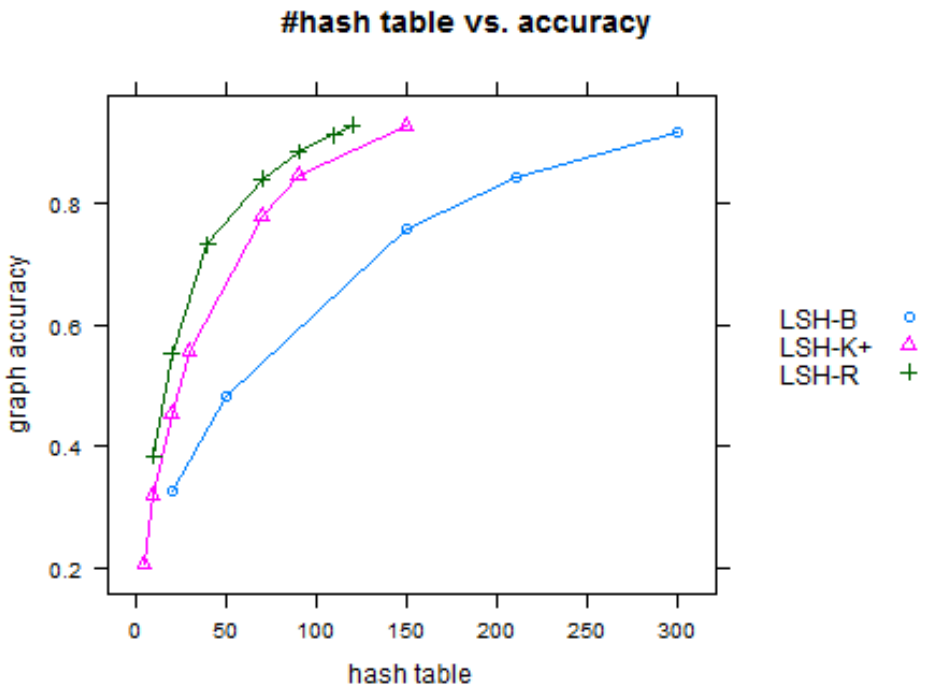
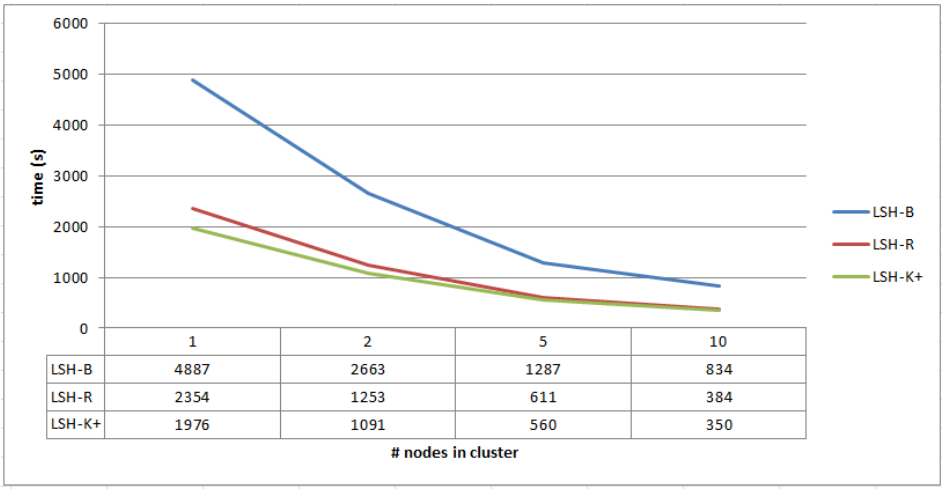


그림 18 뉴욕타임즈 데이터에서 해시테이블 개수와 정확도 비교

5.2.4 클러스터 증가에 따른 영향

본 논문이 제안하는 알고리즘이 클러스터 노드의 변화에 따른 성능의

변화를 확인하고자 무비렌즈 데이터 셋에 대하여 태스크 트래커인 슬레이브 컴퓨터의 개수를 1개 에서 10개로 변화시켜 가며 실험하였다. 그림 19은 태스크 트래커의 개수의 변화에 따른 90% 정확도의 그래프를 생성하는데 걸리는 시간이다. x축은 태스크 트래커의 개수, y축은 소요되는 시간을 의미한다. 태스크 트래커가 증가함에 따라 그래프 생성에 소요되는 시간이 짧아지는 것을 볼 수 있다.



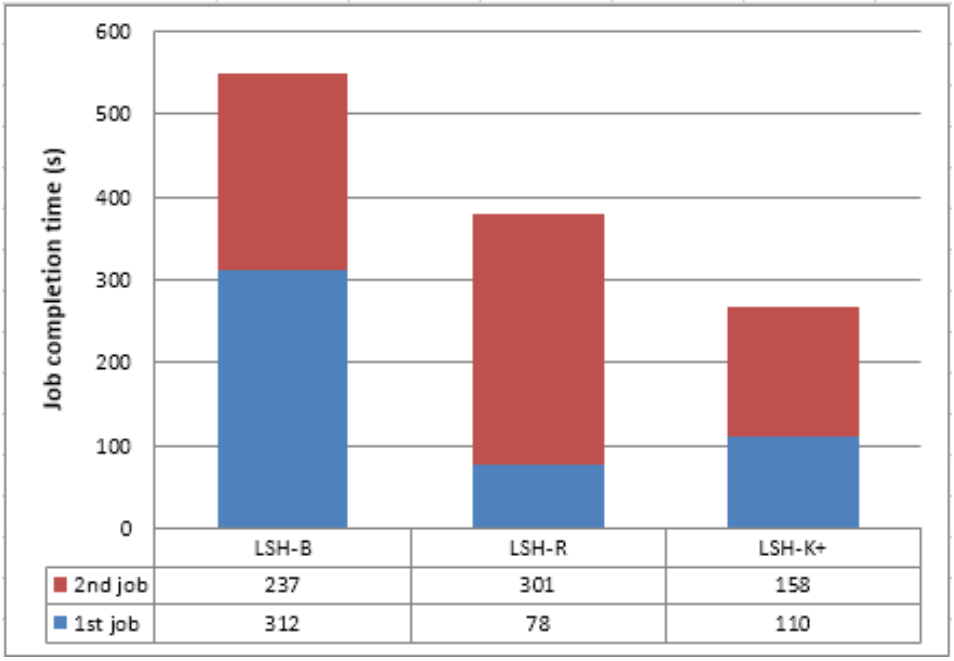
	1	2	5	10
LSH-B	4887	2663	1287	834
LSH-R	2354	1253	611	384
LSH-K+	1976	1091	560	350

그림 19 여러 알고리즘에 대한 scalability 확인

5.2.5 잡 수행시간(job completion time)과 리듀스 셔플 바이트(reduce shuffle bytes)

맵리듀스에서는 인덱스와 데이터를 메시지화하여 보내기 때문에 네트워크 비용을 줄이는 것이 중요하다. 즉, reduce shuffle bytes를 줄이는 것이 중요하다. 그림 20는 뉴욕타임즈 데이터에서 알고리즘 별로 90%의 정확도의 그래프를 생성할 때 각 잡에서 걸렸던 시간과 각 리듀스에서의

셔플사이즈를 나타낸다. LSH-K+알고리즘이 가장 reduce shuffle bytes가 작음을 확인하였다. 그림 21은 무비렌즈 데이터에서 알고리즘 별로 90% 정확도의 그래프를 생성할 때 각 잡에서 걸렸던 시간과 각 리듀스에서의 셔플사이즈를 나타낸다. 뉴욕타임즈 데이터에서의 실험과 마찬가지로 LSH-K+알고리즘이 가장 reduce shuffle bytes가 작다. 4장에서 언급했듯이 LSH-R알고리즘과 비교하였을 때, LSH-K+알고리즘의 경우 추가적인 해시를 하게 된다. 첫 번째 잡에서 LSH-K+알고리즘이 조금 더 수행시간이 길며 reduce shuffle bytes도 더 많다. 하지만, LSH-K+알고리즘은 두 번째 잡에서 더욱 효율적으로 유사도를 계산하기 때문에 두 번째 잡에서 더 많은 수행시간을 단축하여 결과적으로 더 적은 네트워크 비용과 더 짧은 시간에 k-NN그래프를 생성한다.



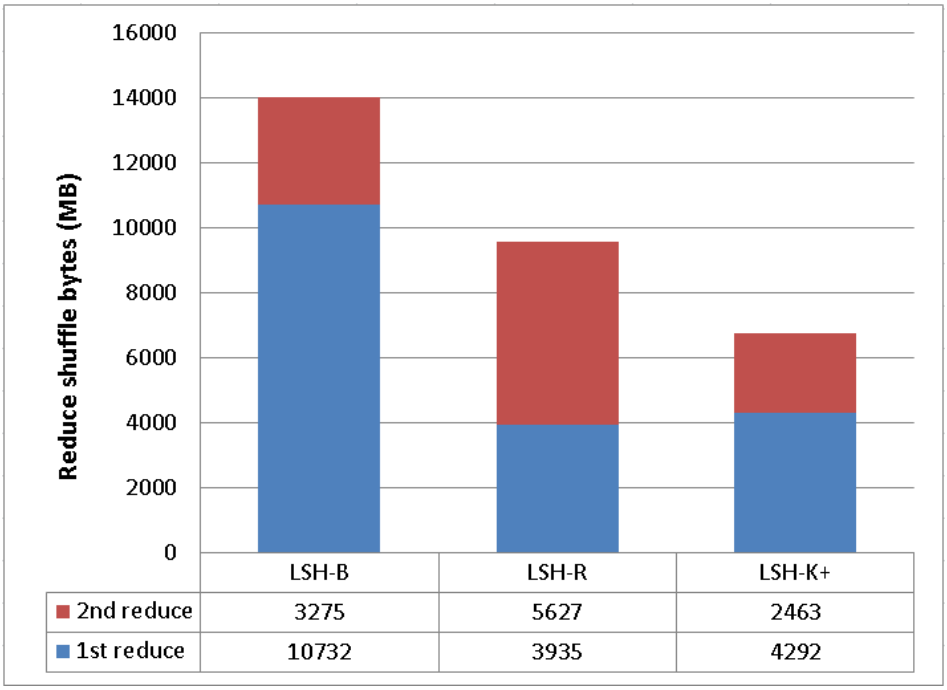
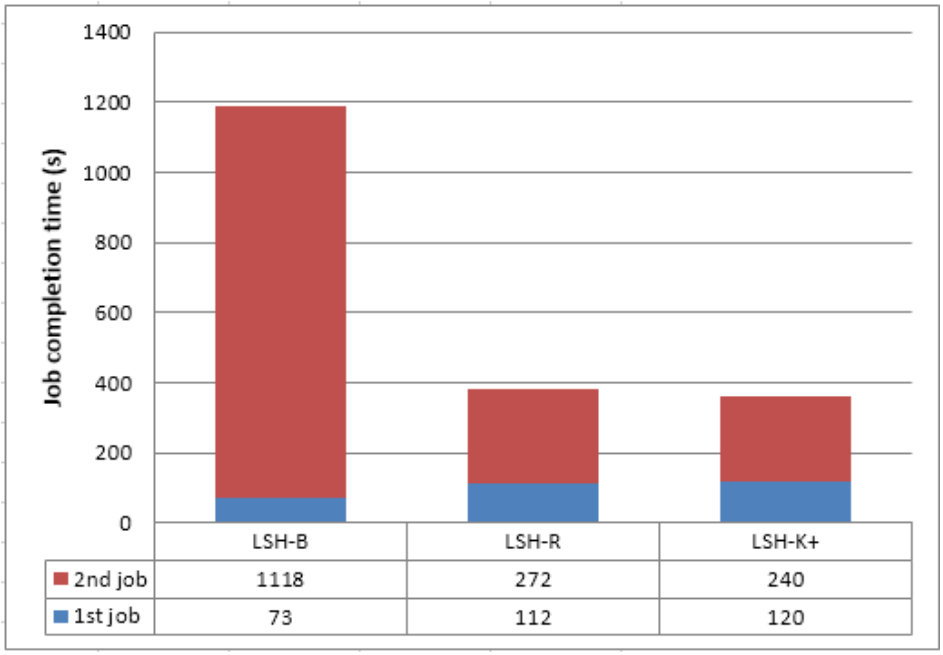


그림 20 뉴욕 타임즈 데이터에서 job completion time과 reduce shuffle bytes



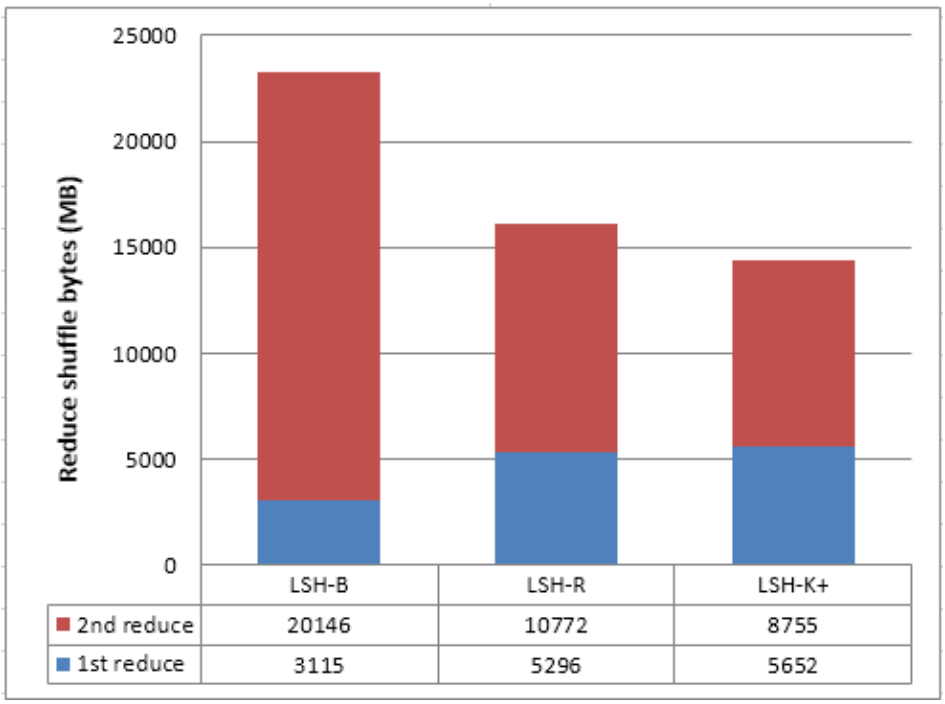


그림 21 무비렌즈 데이터에서 job completion time과 reduce shuffle bytes

5.2.6 맵 수행시간

앞서 실시한 실험에서 알고리즘별 맵 수행시간을 비교한다. 맵리듀스에서 맵은 병렬로 실행되고 모든 맵 과정이 다 끝난 후 리듀스 과정이 시작되기 때문에 모든 맵이 같은 양의 작업을 했을 때가 가장 이상적이다[15]. 하지만, LSH-B의 경우 큰 그룹의 영향 때문에 맵스큐가 일어나기 쉬웠다. 그림 22을 보면 LSH-B의 경우가 맵의 수행시간의 차이가 가장 큰 것을 볼 수 있고, LSH-R의 경우가 맵의 수행시간이 가장 일정한 것을 볼 수 있다. LSH-R알고리즘의 경우가 load balancing이 가장 잘 되고 있다는 것을 확인 할 수 있다.

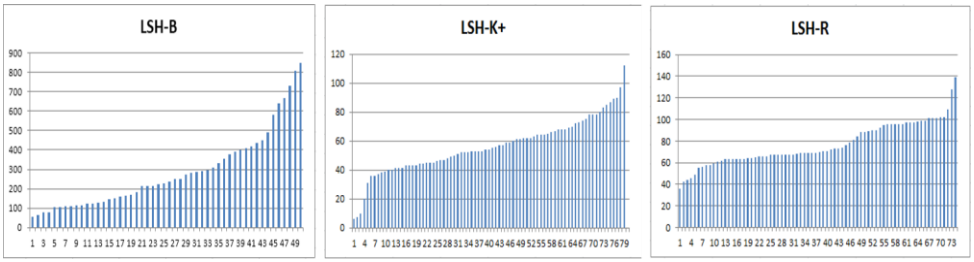


그림 22 알고리즘별 맵 수행시간

제 6 장

결론 및 향후 연구

본 논문에서는 맵리듀스(MapReduce)환경에서의 효율적인 LSH기법 기반의 k-NN그래프 생성 방법을 제시하였다.

본 논문의 방법은 LSH기법인 MinHash로 사용자 또는 신문기사를 작은 그룹으로 나누고, 각 그룹내에서 유사도를 측정한다. 그룹 내에서 brute-force하게 유사도를 계산하기 때문에 유사도가 높은 사용자를 그룹에 속하게 하면서 작은 그룹을 만드는 것이 중요하다. 본 논문에서는 사이즈가 큰 그룹의 재구성방법을 제안 하였다. 그룹의 최대 사이즈를 조절하여 큰 그룹에 속한 일부 사용자에 대해서는 해시 값의 개수를 다르게 하여 그룹을 재구성하였다. 실험 결과 LSH-K+를 사용한 본 논문의 방법이 기존의 방법보다 더 적은 비율의 유사도 측정을 통해 정확도가 더 높은 그래프를 생성하는 것을 확인하였다. 물론, 더 적은 시간에 더 정확도가 높은 그래프를 생성하는 것 또한 확인하였다. 향후 연구로는 더 정밀한 그룹생성을 위한 기법에 대해 연구하고자 하며, secondary sorting 등을 이용하여 구축한 맵리듀스 알고리즘의 최적화 기법을 연구할 계획이다. 또한, 다양한 유사도 계수에 적용 가능한 알고리즘에 대해 연구하고자 한다.

참고문헌

[1] A. Das, M. Datar, A. Garg, and S. Rajaram, "Google news personalization: scalable online collaborative filtering," *Proc. 16th Int. Conf.*, pp. 271–280, 2007.

[2] W. Dong, C. Moses, and K. Li, "Efficient k-nearest neighbor graph construction for generic similarity measures," *Proc. 20th Int. Conf. World wide web - WWW '11*, pp. 577–586, 2011.

[3] M. R. Brito, E. L. Chávez, A. J. Quiroz, and J. E. Yukich, "Connectivity of the mutual k-nearest-neighbor graph in clustering and outlier detection," *Statistics & Probability Letters*, vol. 35. pp. 33–42, 1997.

[4] O. Boiman, E. Shechtman, and M. Irani, "In defense of nearest-neighbor based image classification," in *26th IEEE Conference on Computer Vision and Pattern Recognition, CVPR*, 2008.

[5] Y. Zhang, K. Huang, G. Geng, and C. Liu, "Fast k NN Graph Construction with Locality Sensitive Hashing," *Knowl. Discov. Databases*, pp. 660–674, 2013.

[6] J. Chen, H. Fang, and Y. Saad, "Fast Approximate kNN Graph Construction for High Dimensional Data via Recursive Lanczos Bisection," *J. Mach. Learn. Res.*, vol. 10, no. 2009, pp. 1989–2012, 2009.

[7] Y. Park, S. Park, S. Lee, and W. Jung, "Fast collaborative filtering with a k-nearest neighbor graph," *BigComp*, pp. 92-95, 2014.

[8] J. L. Bentley, "Multidimensional binary search trees used for associative searching," *Communications of the ACM*, vol. 18. pp. 509-517, 1975.

[9] A. Guttman, "R-trees: A Dynamic Index Structure for Spatial Searching," in *Proceedings of the 1984 ACM SIGMOD International Conference on Management of Data - SIGMOD '84*, 1984, pp. 47-57.

[10] R. Weber, H. J. Schek, and S. Blott, "A Quantitative Analysis and Performance Study for Similarity-Search Methods in High-Dimensional Spaces," *Proc. 24th VLDB Conf.*, vol. New York C, pp. 194-205, 1998.

[11] P. Indyk and R. Motwani, "Approximate nearest neighbors: towards removing the curse of dimensionality," in *STOC '98: Proceedings of the thirtieth annual ACM symposium on Theory of computing*, 1998, pp. 604-613.

[12] E. Kushilevitz, R. Ostrovsky, and Y. Rabani, "Efficient search for approximate nearest neighbor in high dimensional spaces," in *STOC '98: Proceedings of the thirtieth annual ACM symposium on Theory of computing*, 1998, pp. 614-623.

[13] L. Li, D. Wang, T. Li, D. Knox, and B. Padmanabhan, "SCENE: a scalable two-stage personalized news recommendation system.," *SIGIR*, pp. 125-134, 2011.

[14] L. Hsieh and G. Wu, "Two-stage sparse graph construction using MinHash on MapReduce," *ICASSP*, pp. 1013-1016, 2012.

[15] Y. Kwon and M. Balazinska, "A study of skew in mapreduce applications," *Open Cirrus Summit*, 2011.

[16] A. Z. Broder, "On the resemblance and containment of documents," *Proceedings. Compression Complex. Seq. 1997 (Cat. No.97TB100171)*, 1997.

[17] R. Szmit, "Locality Sensitive Hashing for Similarity Search Using MapReduce on Large Scale Data," in *IIS*, 2013, vol. 7912, no. LNCS, pp. 171-178.

[18] M. S. Charikar, "Similarity estimation techniques from rounding algorithms," in *Proceedings of the thiry-fourth annual ACM symposium on Theory of computing - STOC '02*, 2002, pp. 380-388.

[19] A. Shrivastava and P. Li, "In Defense of MinHash Over SimHash," *JMLR W&CP*, vol. 33, pp. 886-894, 2014.

[20] "Apache Hadoop." [Online]. Available: http://hadoop.apache.org/.

[21] J. Dean and S. Ghemawat, "MapReduce : Simplified Data Processing on Large Clusters," *Commun. ACM*, vol. 51, pp. 1-13, 2008.

[22] J. Ekanayake, S. Pallickara, and G. Fox, "MapReduce for data intensive scientific analyses," in *Proceedings - 4th IEEE*

International Conference on eScience, eScience 2008, 2008, pp. 277-284.

[23] Z. Yang, W. Oop, and Q. Sun, "Hierarchical non-uniform locally sensitive hashing and its application to video identification," *ICIP*, pp. 743-746, 2004.

[24] "MovieLens." [Online]. Available: http://grouplens.org/datasets/movielens/.

[25] "NYTimes news articles." [Online]. Available: https://archive.ics.uci.edu/ml/datasets/Bag+of+Words.

부록

I. 하둡 설정

본 논문의 실험에서 사용된 하둡 설정은 다음과 같다.

A. mapred-site.xml

```
<configuration>
    <property>
    <name>mapred.job.tracker</name>
    <value>onix1:9101</value>
    </property>
    <property>
    <name>mapred.tasktracker.map.tasks.maximum</name>
    <value>3</value>
    </property>
    <property>
    <name>mapred.tasktracker.reduce.tasks.maximum</name>
    <value>3</value>
    </property>
    <property>
    <name>mapred.reduce.tasks</name>
    <value>28</value>
    </property>
    <property>
    <name>mapred.jobtracker.completeuserjobs.maximum</name>
    <value>5</value>
    </property>
    <property>
    <name>mapred.job.tracker.jobhistory.lru.cache.size</name>
    <value>1</value>
    </property>
</configuration>
```

B. hdfs-site.xml

```
<configuration>
<property>
<name>dfs.replication</name>
<value>3</value>
</property>

<property>
<name>dfs.data.dir</name>
<value>/home/hsoh/dfs/data</value>
<final>true</final>
</property>

<property>
<name>dfs.name.dir</name>
<value>/home/hsoh/dfs/name</value>
<final>true</final>
</property>

<property>
<name>dfs.name.edits.dir</name>
<value>${dfs.name.dir}</value>
</property>
</configuration>
```

II. 사용 데이터 셋

데이터 셋은 Node ID와 item IDs는 <tab>으로 구분 되고 Item ID는 <space>로 구분된다. 무비렌즈 데이터의 경우 Node ID는 User ID이며, 뉴욕 타임즈 데이터의 경우에는 Node ID는 Doc ID이고 Item ID는 Word ID이다.

A. 무비렌즈 데이터 셋의 일부

```
1	122 185 231 292 316 329 355 356 362 364 370 377 420 466
480 520 539 586 588 589 594 616
2	110 260 590 1210
3	110 151 213 1148 1246 1252 1564 1597 1674 3408 3684 4535
4677 4995 5527 6377 6539 7153 8529 8533 8783 27821
4	34 110 150 153 161 165 266 316 317 329 364 410 480 500 586
587 588 589 590 592 595
5	30 32 47 52 111 230 235 249 307 326 334 348 412 446 495
509 527 532 538 541 562 593 608 778 858 903 912 919 920 923 926
969 1041 1046 1073 1080 1094 1096 1103 1104 1172 1183 1199 1206
1207 1219 1221 1225 1230 1235 1244 1247 1258 1280 1295 1300
6	32 260 349 457 858 1193 1196 1197 1198 1264 1277 1304 1396
1527 1580 1584 1629 1653 1748 2028 2396 2571 2628 3578 3753 3755
3994 3996 4299
7	32 50 101 599 608 800 899 903 904 908 912 913 920 923 930
942 951 1086 1148 1196 1207 1212 1219 1234 1244 1245 1248 1252
1254 1256 1260 1266 1270 1276 1283 1284 1288 1304 1333 1344 1348
1387 1517 1617 1732 1748 2186 2203 2206 2391 2395 2648 2762 2791
2936 3176 3334 3365 3435 3467 3703 3730 3994 4226 4327 4406 4420
4432 4975 5292 5294 5300 5388 5502 5505 5528 6273
8	6 19 31 32 36 47 50 70 104 141 145 153 163 165 170 172 215
216 240 253 260 288 290 292 303 316 324 328 338 344 353 368 372
384 407 435 442 457 471 479 489 514 520 522 527 541 543 548 552
589 592 593 608 610 637 678 736 778 780 784 785 799 842 849 934
996 1020 1036 1037 1060 1061 1064 1079 1080 1083 1089 1093 1100
1125 1127 1129 1136 1148 1196 1197 1198 1200 1210 1214 1215 1220
1234 1240 1253 1262 1265 1268 1270 1285 1291 1320 1321 1347 1358
1370 1373 1391 1395 1396 1409 1425 1429 1466 1515 1517 1527 1573
1580 1588 1591 1598 1611 1616 1617 1625 1639 1644 1653 1658 1663
1665 1673 1676 1687 1690 1704 1717 1722 1732 1748 1754 1792 1805
1831 1883 1918 1920 1923 1954 1969 1971 1973 1974 1994 2000 2001
2002 2003 2004 2005 2011 2012 2021 2026 2028 2058 2072 2105 2115
2124 2140 2161 2167 2193 2231 2247 2253 2278 2294 2322 2329 2335
2355 2387 2393 2405 2406 2431 2433 2447 2478 2490 2496 2498 2540
2541 2542 2571 2572 2598 2600 2606 2617 2618 2640 2641 2683 2701
2713 2719 2735 2762 2793 2802 2805 2808 2841 2858 2889 2893 2913
```

B. 뉴욕 타임즈 데이터 셋의 일부

```
1     413 534 2340 2806 3059 3070 3294 3356 4056 4930 5255 6888
6946 6974 7296 7402 7405 7409 7544 7790 9085 9385 9959 9983 10126
10474 10787 11762 12610 12961 13359 13992 14255 14753 14815 14852
15503 15713 15886 16253 16385 16581 16743 16852 17654 17660 17820
18072 18200 18353 18566 18704 18990 22127 22128 22147 22291 22313
22872 23507 24489 24858 25611 25723 25724 25952 26114 26404 28051
28409 28410 29167 29176 29363 29609 30679 31013 31440 31586 31588
31745 31748 31943 32557 33023 33472 33946 34463 34498 34563 34784
34892 35491 35495 35542 37904 37961 38055 39124 39139 39144 39349
40235 40385 40479 41055 41062 41161 42526 42861 43569 43808 43981
43982 49517 49518 51749 52088 54197 56963 58669 63370 68127 78319
80416 82034 83139 83177
2     442 623 1020 1698 1700 1706 1894 1938 2006 2042 2188 2197
3193 3356 4203 4353 5244 5249 5796 5921 6003 6005 6111 6837 6870
7633 8428 8468 9316 9606 9746 10480 11334 12261 12263 12420 13153
13407 13850 14110 14631 14911 15722 15953 16287 16698 17981 18442
18536 20954 20957 21757 22128 22177 22442 22799 23507 23955 24007
24086 24218 24489 25701 25882 25896 25961 26310 26404 26406 26411
26693 27641 27798 28028 28543 28566 29718 29968 30008 30347 30417
30571 30797 30808 31615 31808 32113 32557 34414 35102 36035 36545
36576 37242 37607 38018 40691 40787 41050 42025 42329 43107 43984
44053 44382 90162 92098 99088
3     59 396 400 404 1002 1224 1630 2008 2331 2806 3017 3294 3769
4251 4322 4729 4732 4850 5065 5273 5367 5479 5560 5712 5713 5918
5922 6005 6111 6128 6140 6249 6265 6354 6435 6771 6889 7270 7468
7507 7681 7682 7692 8858 9085 9298 9377 9545 9643 10423 10510
10819 11403 11922 11956 12261 12726 12966 13177 13205 13553 13684
13966 14282 14304 14382 14395 14570 14592 14626 14668 14722 14881
14929 14999 15179 15402 15479 15900 16137 16160 16210 16253 16336
16776 17122 17127 17209 17726 18100 18118 18206 18284 18529 18788
18793 18806 18828 18832 18980 19027 19227 19824 21227 21238 21560
21924 22095 22142 22177 22442 22537 22645 23204 23237 23622 23744
23947 23969 24013 24269 24385 24788 25397 25600 25675 25728 25771
25896 25952 25956 26240 26703 26846 26898 27029 27141 27532 27673
28025 28051 28076 28295 28303 28459 29110 29175 29363 29399 29524
29654 29718 30172 30298 30354 30426 30496 30627 31002 31274 31588
32175 32566 33256 33550 33686 33724 33971 34388 34638 34770 34793
34837 34847 34853 35169 35176 35307 35560 35570 35579 35884 35946
35963 36009 36310 36915 36952 36987 36989 37404 37667 37851 37968
37975 38024 38056 38237 38321 38492 38516 39196 39414 39487 39663
39665 39720 39745 39785 39838 39844 39889 40144 40211 40460 40820
40931 41657 42357 43020 43223 43238 43244 43386 43395 43478 43545
43569 43676 43758 43985 44079 44239 44423 46141 50500 51342 52404
54752 58389 58927 64816 69136 72664 73577 75752 76689 78205 81862
82418 85291 88770 89126 89570 91021 91898 92098 92343 93303 95228
```

III. 실험 아웃풋

아웃풋의 형식은 node ID <tab> node ID <space> 자카드 계수이다.

A. 무비렌즈

```
1	26370 0.37037037037037035
1	21281 0.3333333333333333
1	59001 0.3333333333333333
1	53056 0.3235294117647059
1	25718 0.3170731707317073
1	52300 0.3157894736842105
1	46741 0.3103448275862069
1	52212 0.30434782608695654
1	67747 0.30303030303030304
1	68029 0.30303030303030304
1	16280 0.3023255813953488
1	53256 0.3
1	64389 0.3
1	30683 0.2972972972972973
1	64224 0.2972972972972973
1	6346  0.29411764705882354
1	19301 0.29411764705882354
1	28975 0.29411764705882354
1	67740 0.29411764705882354
1	3666  0.2903225806451613
1	10300 0.2903225806451613
1	69417 0.2903225806451613
1	36767 0.2894736842105263
1	22207 0.2857142857142857
1	30856 0.2857142857142857
1	33326 0.2857142857142857
1	61188 0.2857142857142857
1	23119 0.2826086956521739
1	55605 0.28205128205128205
1	27958 0.28125
1	32290 0.28125
1	59224 0.28125
1	15946 0.27906976744186046
1	31035 0.2777777777777778
1	68430 0.27586206896551724
1	64856 0.275
1	7167  0.2727272727272727
1	15955 0.2727272727272727
1	16441 0.2727272727272727
1	27249 0.2727272727272727
1	26681 0.2727272727272727
1	61089 0.2727272727272727
```

B. 뉴욕 타임즈

1	18299	0.16595744680851063
1	16173	0.15789473684210525
1	6838	0.1575091575091575
1	19789	0.1552511415525114
1	1787	0.15311004784688995
1	18640	0.15234375
1	1041	0.1519434628975265
1	6859	0.14957264957264957
1	11270	0.14957264957264957
1	16542	0.14957264957264957
2	4320	0.05434782608695652
2	6371	0.05241935483870968
2	8450	0.05223880597014925
2	698	0.05084745762711865
2	1961	0.049707602339181284
2	462	0.049586776859504134
2	452	0.049586776859504134
2	1093	0.049586776859504134
2	17000	0.049429657794676805
2	17010	0.049429657794676805
3	25	0.11410788381742738
3	7	0.0967741935483871
3	232	0.0872865275142315
3	11	0.08353808353808354
3	27	0.08041237113402062
3	17328	0.07581967213114754
3	7150	0.075
3	17230	0.07407407407407407
3	10649	0.07392197125256673
3	126	0.07360861759425494
4	26	0.13109756097560976
4	167	0.12637362637362637
4	166	0.12637362637362637
4	103	0.12462908011869436
4	104	0.12462908011869436
4	123	0.12462908011869436
4	1593	0.11377245508982035
4	1592	0.11377245508982035
4	175	0.1087866108786611
4	17772	0.09716599190283401

Abstract

An Improvement in K-NN Graph Construction with Locality Sensitive Hashing on MapReduce

Inhoe Lee

Department of Computer Science and Engineering

The Graduate School

Seoul National University

The k nearest neighbor (k-NN) graph construction is an important operation with many web related applications, including collaborative filtering, similarity search, and many others in data mining and machine learning. Despite its many elegant properties, the brute-force k-NN graph construction method has computational complexity

of $O(n^2)$, which is prohibitive for large scale data sets. <Key, Value> based distributed frameworks, MapReduce is gaining increasingly widespread use in applications that process large amounts of data. Based on the divide-and-conquer(two-stage) strategy, we engage the locality sensitive hashing technique which is used for high-dimension and sparse data to divide users into small groups, and calculate similarity using brute-force method on MapReduce. Specifically, generating candidate group stage is important since brute-force calculation is performed in following step. In this paper, we proposed an efficient algorithm for approximating k-NN graphs by re-grouping candidate group using hierarchical LSH. Experimental results show that our approach is more effective than existing method in aspects of graph accuracy and scan rate.

Keywords : Big Data, Hadoop, MapReduce, k-NN Graph Construction, Locality Sensitive Hashing, MinHash

Student Number : 2013-20855